Analysis of Application Behavior

During Fault Injection

Paul L. Springer

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, 168-522
Pasadena, CA 91109

Abstract: How well will an application run in an environment in which memory and processor bits may be changed because of exposure to radiation? This paper looks at the behavior of a single application when register, heap, and code space are injected with faults. An analysis is done of the program's response to injected faults and generalizations are drawn with regards to some of the characteristics of a program which make it more or less susceptible to producing incorrect results when faults are injected.

1. Introduction

The Remote Exploration and Experimentation (REE) project at JPL is developing an embedded cluster that uses COTS (Commercial-Off-The-Shelf) components, for use on-board spacecraft [1], [2]. There is an advantage in using COTS hardware instead of radiation-hardened components, because the process of radiation-hardening a device takes time, and as a result, radiation-hardened components are a generation or more behind what is available in the commercial market.

Of course the disadvantage of using COTS components is that they are susceptible to radiation effects. One of the goals of the REE project is to determine how radiation exposure will affect the operation of scientific applications, and whether radiation effects can be mitigated by means of strategies such as the use of Algorithm-Based Fault Tolerance (ABFT) techniques [3]. The first step in this process is to understand how an application will fare on its own in a radiation environment. Understanding this will provide valuable information on what steps can be taken to improve performance in a radiation environment.

The radiation environment the REE project is considering initially is the one to which satellites are exposed while in low earth orbit (LEO) or geosynchronous orbit (GEO). The project developed a radiation fault model for the hardware we expect to use, in these environments. According to the model, we would expect to see less than 20 faults per hour per node, assuming nominal solar activity [4]. The vast majority of these faults (over 99.9%) we expect to be transient errors, in the form of single event upsets (SEU's).

Because the REE system is being designed primarily for support of science applications, as opposed to mission-critical or spacecraft control functions, occasional resets and processing delays as a result of SEU's are acceptable. Infrequent program crashes or hangs are tolerable because they are easily detected, and the program can be restarted. Also allowable are very slight differences in the output, equivalent to the

differences that might result from porting the program to an alternate platform, or differences that might result from small amounts of noise in the input. For this test, differences in 50 or fewer of the 256K segmented pixels output by the program were considered acceptable. In contrast, an SEU that produced an error resulting in output differing in more than 50 positions is considered an unacceptable error. These unacceptable errors are the kind from which we want to protect the application, through the use of ABFT libraries or other means.

2. The Application

The application we are presently studying is one produced by the Mars Rover Science Team. The application is called segment_rocks, and does image segmentation by first performing texture analysis on a landscape image, and then clustering those outputs. A block diagram of the program's data flow is shown in Figure 1.

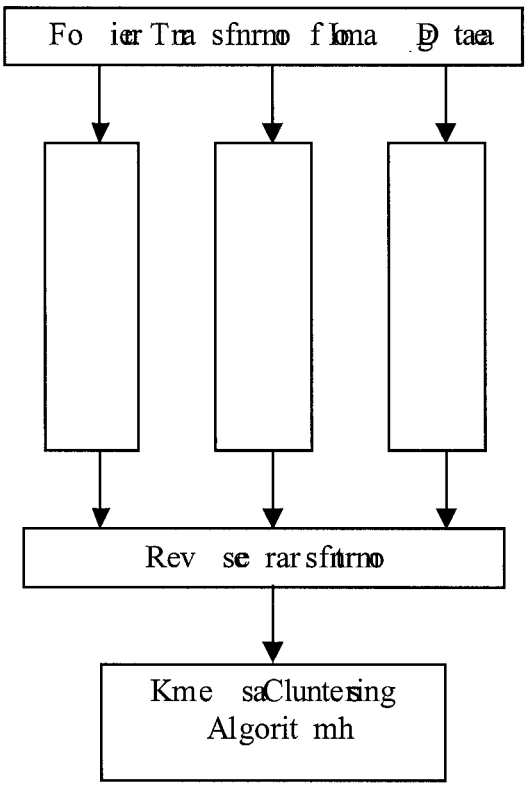

Figure 1. Block diagram of application

The program first reads the image file from the disk. It then calls the FFTW library to transform the data. The transformed data is then run through a Gabor filter which highlights texture and orientation features. A reverse transform is then performed on the filtered data. The filtering and reverse transformation steps are done three times in sequence for different filters. The kmeans routine then clusters the pixels in the image

based on the values returned by the three reverse transformations. The resultant output is a 512 x 512 "labels file" which maps each pixel of the input to an integer. This integer has a range determined by a program parameter used to request the number of clusters. For this test, the output integers ranged in value from one to three.

Our testbed is comprised of PowerPC 750 processors, running at 366 MHz. Each processor has 32 KB of instruction cache, 32 KB of data cache, and 1 MB of L2 cache, as well as 64 MB of main memory. Although the program can be run in parallel on more than one processor, for the purposes of this analysis only the single node configuration was used. To aid in the analysis of program behavior, no compiler optimization flags were set.

As previously mentioned, one of the program parameters (the k parameter) governs the number of clusters into which the program will segment the image. The program was run with this parameter set to three. The program I/O was not taken into consideration either in the timing or the fault injection, partly to simplify the analysis, and partly because we are not yet sure how our project will handle I/O--it may be written directly to solid state memory. Fault injection was done only into the application space--no faults were injected into the O/S.

As a first step in the analysis, the different parts of the program were timed. Program initialization took 0.1 seconds and the forward transform took 0.5 seconds. The forward transform is done on a 512 x 512-pixel array, with each pixel containing double precision data. Each of the three Gabor filtering steps lasted 0.35 seconds, and each of the three reverse transforms ran in 0.5 seconds. The reverse transformed data is then normalized, scaled, and converted from double to single precision in about 0.3 seconds for each of the three filtered outputs. The kmeans routine took about 90% of the time, lasting 30.5 seconds. The total program run time was 34.5 seconds. These times and the analysis that follows depend on the exact parameter settings and program input that was used. Different input or settings could cause the program to follow other paths, which would alter its response to an SEU. To simplify the analysis here, the assumption is made that the program only runs a single time once it is loaded into memory.

3. Fault Injection

When the REE test team began automated fault injection, their results indicated that when the application was injected with a fault, it produced wrong answers less than 10% of the time. This prompted an effort to determine exactly how the program was affected when hit by an SEU, and why it seemed robust enough to still produce correct answers.

For the purposes of this research, our own homegrown fault injector, by the name of SWIFI, was used. SWIFI works by using the ptrace facility to inject faults into user designated areas, including the code, registers, stack, heap, and data. A total of 60 runs were completed. Twenty runs forced a fault into register space, 20 into the heap, and 20 into the code. This was a Monte Carlo simulation, with the locations of the injected faults within heap, code, and register space being randomly chosen by SWIFI. In each case, only one SEU was simulated for each time the program was run.

Heap Injections

Table 1 shows the results of injecting faults into the program's heap. In 13 out of 20 cases, the fault was injected into an array that was not active at the time; consequently, these faults caused no change in program behavior. Even when the fault was injected into an area of the heap that was in use, the program proved to be surprisingly robust, only once resulting in an unacceptable error in the output. This brings up the question of what attributes of the program made it this fault-tolerant, and whether other programs can be modified to give them some of the same characteristics.

Injection Location	No Error	Acceptable Error	Unacceptable Error	Crash or Hang
Previously used memory	11	0	0	0
Not yet used memory	2	0	0	0
In-use memory	4	2	1	0

Table 1. Outcomes of SEU's in Heap

First we look at the way memory is allocated and used by the segment_rocks application. Of the entire heap space, 99.8% is allocated into 7 large arrays, each in size between 1 and 4 MB. Looking at when these arrays are in use will give us the information we need. Figure 2 is a chart based on this information. It is drawn in the time domain, and shows at what times during program execution the seven large arrays contain data that is to be used. If the array has not yet been filled with data, or has data in it that is no longer needed, it does not count against the amount of heap space in use at that time.

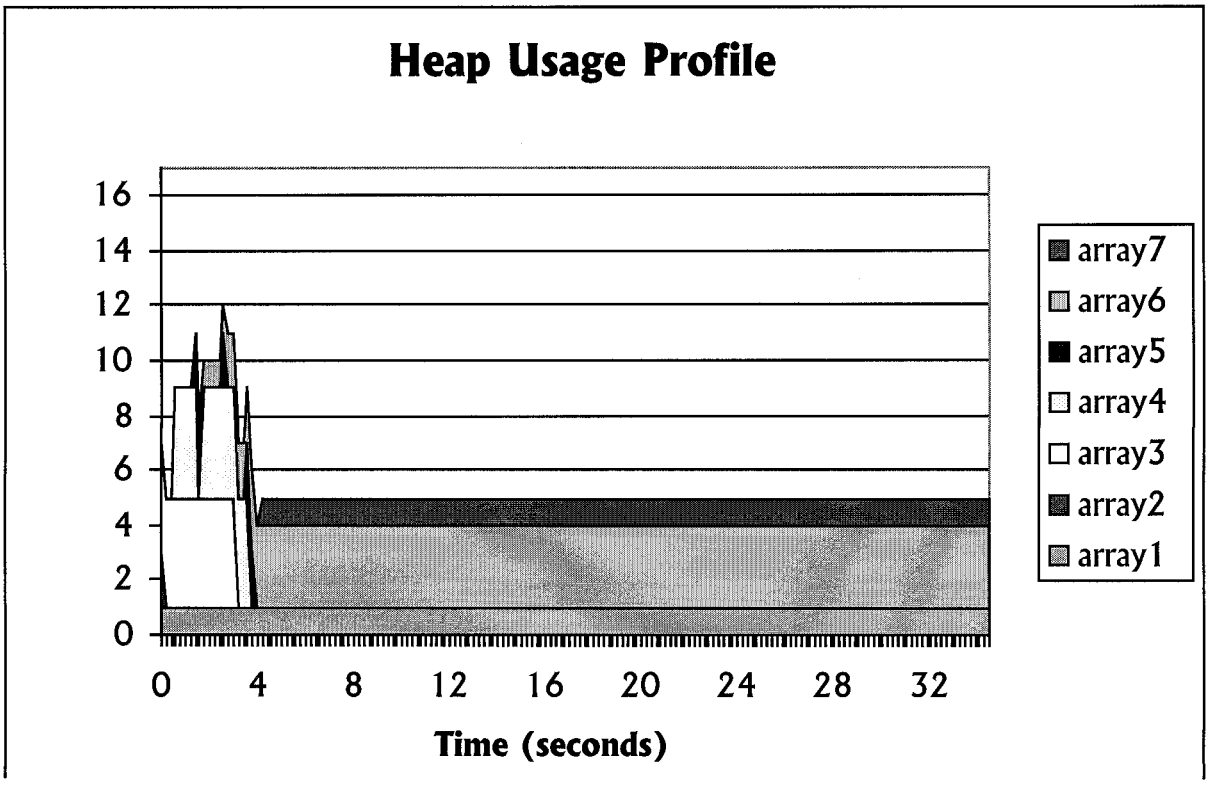

Figure 2. Heap Usage Profile

In the profile, the kmeans routine starts at about the 4-second mark. From that point on, the program uses only three arrays, which contain a total of 5 MB. By calculating the area of the chart, one can determine that on the average only 5.7 MB is in active use over the course of the program run, even though a total of 17 MB is allocated. Consequently, a single fault injection into the heap has only a 34% chance of affecting memory that is in use at the time.

Of course injecting just a single fault during the time the program is running is not representative of what will actually happen in space. Assuming an environment where the level of radiation is fairly constant, and a particular hardware configuration, the frequency with which SEU's affect the program RAM is proportional to the amount of RAM used and the length of time it is in use [5]. Therefore the important measure of a program's susceptibility to SEU's in this environment can be expressed in units of MB x seconds. The heap space of this program can be characterized as having a MB-seconds value of 197.

The measure of a program's MB-seconds for its heap is useful in various ways. It permits the comparison of runs of the same program with different inputs, as well as comparison of two different programs. For example, program A might overall allocate more memory than program B, but that information alone is not enough to determine which program is more susceptible to SEU's in the heap space. One must also take into account the amount of time during which this memory is actually in use, and that is precisely what the MB-seconds figure does.

The next part of this analysis can be simplified by defining the term vulnerability. An area or array in the heap space can be said to be **vulnerable** at a particular point in time if it contains data that will be used later in the program execution. Seven of the runs resulted in fault injections that were made in arrays that were vulnerable at the time of injection. Of these seven, four resulted in no errors, two had acceptable errors, and only one had an unacceptable error.

The relative immunity of the program to faults injected into vulnerable data can be explained by looking at how the input data is used in the program. Essentially, segment_rocks extracts information from the input data. The amount of input data is physically represented by a 512 x 512 pixel file, with each pixel containing 8 bytes of information, for a total of 16,777,216 bits of input information. By the time the kmeans routine begins, those bits have resulted in 25,165,824 bits of information out of which kmeans extracts 524,288 bits of output information. For the purposes of this analysis, we'll call the ratio of input bits to output bits the **data condensation factor**. In the kmeans stage of this program (where the bulk of the time is spent, and the bulk of the injections occur) the data condensation factor is 48. Though there is no direct or simple relationship between the data condensation factor and the chances of an SEU affecting the output, a value of 48 here indicates that it is not too surprising for errors in the input data to go unnoticed. In fact, the only injection that produced an unacceptable error ended up not in the data, but in a large (1 MB) array of pointers that was used to access the data array. This is probably a good argument to try to avoid large pointer arrays,

either by restructuring the data array, or by modifying the access algorithm for the data array so that a pointer array is unnecessary. Such changes must be done in light of the fact that the longer a program runs, the higher the chance it may be hit with an SEU.

Register Injections

One might think that faults injected into registers would have a more dramatic effect on program behavior, but this was not so. In all of the 20 runs with register fault injection, the program produced correct results. Others have also observed similar results [6], [7]. An examination of the disassembled code produced by the debugger showed that in each of the 20 cases, the register with the injected fault was not in use at the time. This finding in turn led several questions. Was it pure coincidence that SWIFI happened to pick an unused register each time it injected a fault, or did it indicate a low level of register usage by the application? If the program did not use many registers, why was that?

Other research has offered explanations for why faults may not always affect application behavior [7], [8]. In Koga et al [6], a description is given of the duty cycle of each register--the percentage of time during which the register contents are vulnerable to an SEU. The previous discussion of heap vulnerability is an extension of the idea of register vulnerability. As was the case with heap arrays, a register is not vulnerable if it has not yet been loaded with data, nor is it vulnerable if the data in it no longer has any use. This concept could be more clearly illustrated if we could see a profile of register usage, similar to the heap usage profile shown previously. But it is fairly difficult to produce the information necessary to create the profile without some kind of processor simulator, as described in [9], [10]. What can be done without too much effort is to build a tool that looks at register usage not in the time domain, but in the code address domain. A Perl tool by the name of Regprof was built for this purpose. Regprof looks at assembly code to determine which instructions use which registers. It then produces a file that contains a count of how many registers are in use at each program address. The plot of that output for the kmeans routine is shown in Figure 3. This plot does indeed show that register usage is very low. But because the plot does not show register usage in the time domain, it could be misleading, if for example kmeans spends most of its time in a loop around the point where 8 registers are in use. However, additional analysis of the kmeans routine showed that it spent the bulk of its time in the area of code shown between the arrows on the plot, between instructions 316 and 533. The code in this area uses about 4-5 registers, the same number of registers on the average as the entire kmeans routine. So we can conclude that on the average 4-5 registers are vulnerable at any one time. With the PowerPC 750 having a total of 64 general purpose and floating point registers, the expected chance of hitting a register while it has useful data in it is the same as the combined duty cycle of the registers, about 7%.

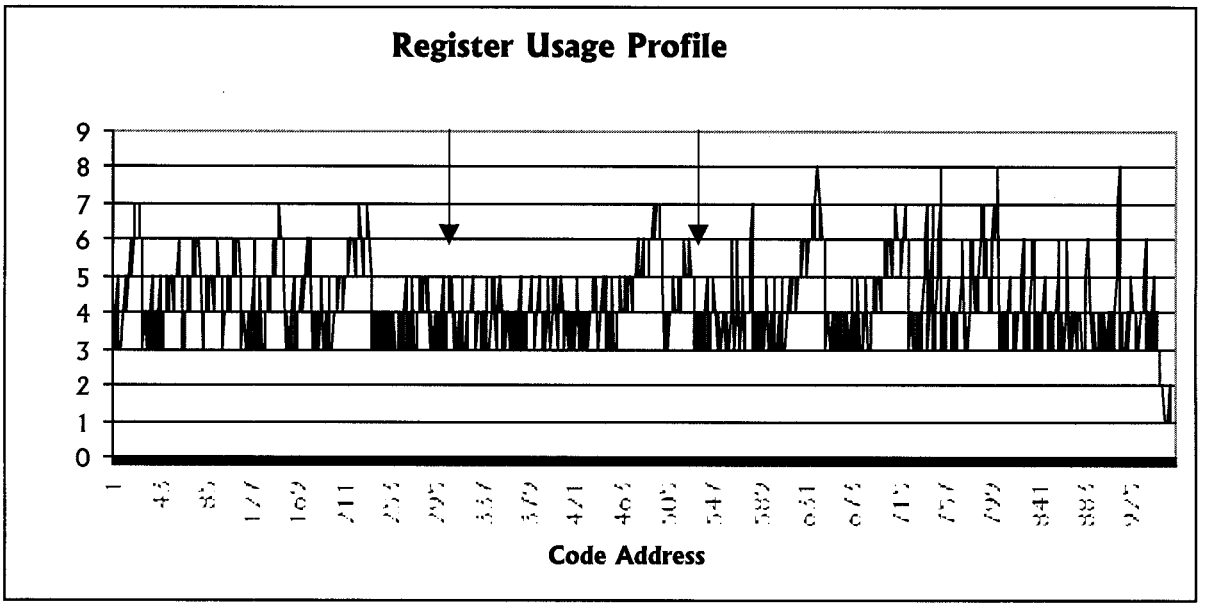

Figure 3. Register Usage Profile for kmeans

It is in the area of register usage that the decision not to use the compiler optimization flags has the largest effect. When the -O flag is used to compile this program, the register usage level climbs so that an average of 14-15 registers are vulnerable over the code space. Although complete analysis of program behavior when compiled with the -O flag has not been completed, 20 runs of the program while injecting faults into the registers resulted in four runs that produced some errors, but none of these fell into the "unacceptable" category. One of the four resulted in a segmentation violation; one produced a hang, and the other two produced errors too small to show up in the output.

This brings up the idea of a tradeoff: if program performance is not seriously compromised, it might make a program more robust against SEU's to compile it without register usage optimization. This would force the program to load a fresh copy of the data into the register each time, rather than relying on old and possibly corrupted data already there. This strategy could be particularly desirable if memory is protected by Error Detection and Correction (EDAC) methods.

An example from the segment_rocks program can illustrate this point. When the optimization flag is used, one of the registers is dedicated to hold a value that is useful in converting integers to floating point values. This conversion is done infrequently in the program. If the program instead loaded this conversion value from EDAC-protected memory when it was needed, instead of leaving it in the register constantly, the program run time would not be noticeable affected, but register reliability would increase because the duty cycle of this register would be lowered.

Code Injections

When faults were injected into the code segment during 20 separate runs, not a single error or change in program execution occurred. In 18 of these runs, a fault was injected into part of the code that at the time of fault injection was no longer needed. In one other run a fault was injected into a branch of code that was not taken. On only one of the 20 runs was a modified instruction actually executed. In that case, the injected fault changed an immediate value loaded into the register that was used as a flag. The change in value did not affect the flag test.

The primary explanation for why so many faults were injected into unused code has to do with the program flow. Most of the individual instructions in the program are executed in the first few seconds. The kmeans part of the program that uses the bulk of the time, uses only a very small amount of code in proportion to the rest of the program. This means that most of the time when a fault is injected, the program will be in the kmeans routine at the end of the code, but the fault will be injected into part of the code that was previously executed.

This description can be simplified by again extending the term "vulnerability". In this case, an instruction in the code space is said to be vulnerable at a given point in time if the processor will execute that instruction before the program finishes. At the beginning of the program the entire code space is vulnerable, and at the end none of it is. Using this concept of code vulnerability, a plot can be drawn showing the amount of vulnerable code as the program executes. This is similar in concept to the heap usage profile of Figure 2, except that the code usage profile will always be a non-increasing function.

It is not easy to determine a complete code usage profile for the segment_rocks program, but the worst case profile shown in Figure 4 is illustrative. At the beginning of execution the entire 305 KB of program space is vulnerable, and at the 34.5 second mark when the program terminates, the vulnerability drops to 0. The kmeans routine is initiated at the 4 second mark, and at that point only 4K of code is vulnerable. Given those constraints and the fact that the vulnerability function is non-increasing through time permits determination of the worst case profile as shown. Note there is an assumption here that the program execution is begun immediately after it is loaded into memory.

With this worst case code usage profile calculated, it is possible to evaluate the probability of a fault injected into code space actually affecting an instruction that will be executed. Dividing the area under the graph by the product of the size of the code space and the total execution time gives a 13% probability (worst case) that an injected fault will end up in vulnerable code.

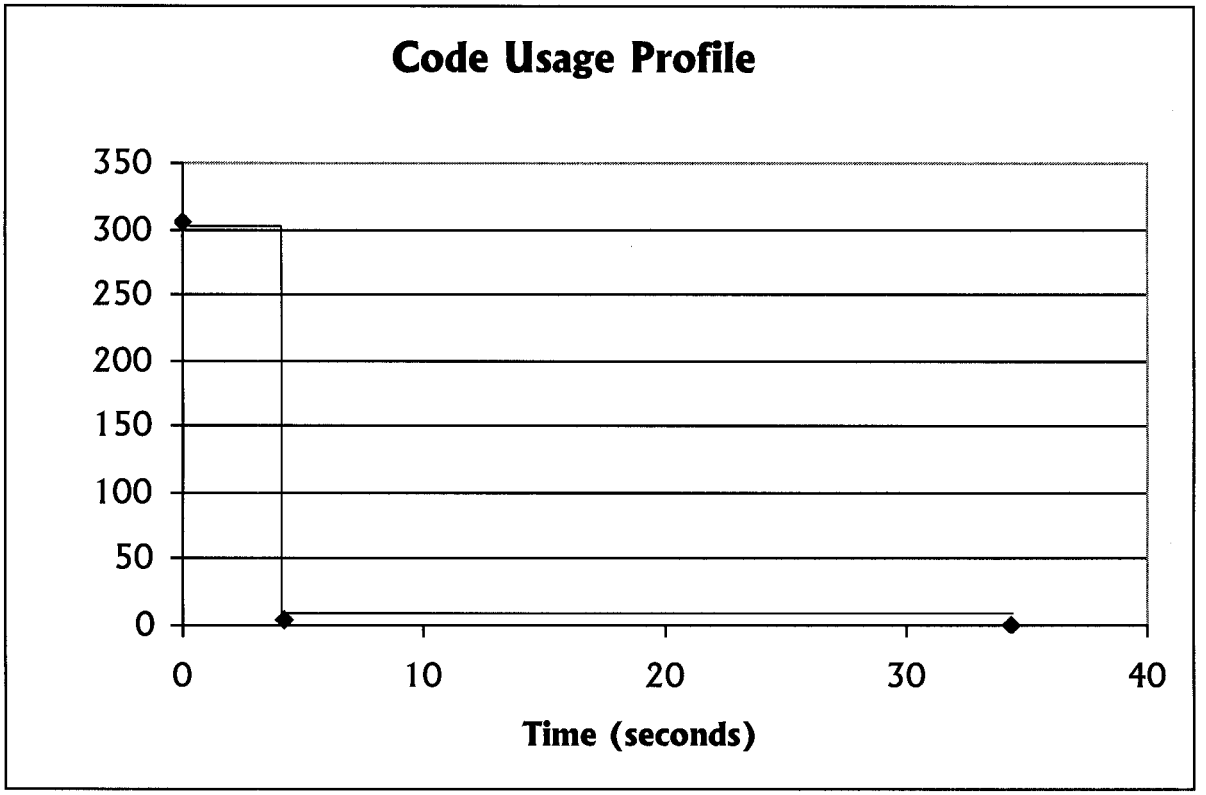

Figure 4. Worst Case Code Usage Profile

It is useful to try to characterize the code usage profile with a single number that will indicate how vulnerable a particular program run is to SEU's that occur in the code space. Such a number will permit comparison with other programs, or even other runs of this program with different input. (It must be pointed out that all of the usage profiles can change with different program parameters or input data.) Similar to the discussion in the section on heap fault injections, the chances of an SEU causing a fault in vulnerable code varies directly in proportion to the amount of vulnerable memory and the period of time during which it is vulnerable. In this case the worst case number is simply the area under the graph, which comes to 1.342 MB-seconds.

Our testbed system, like most computer systems, uses the same memory for heap space and code space. This fact allows the MB-seconds figures for code and heap space to be used in similar ways. For example, we can compare the code vulnerability with the vulnerability of the heap to SEU's, previously calculated to be 197 MB-seconds. From this it is evident that an SEU is far more likely to affect a vulnerable area in the heap than in the code.

The two measures can also be added to obtain a total vulnerability figure across both heap and code memory. This in turn allows evaluation of techniques that trade off one kind of vulnerability for the other. For the sake of illustration, suppose an ABFT strategy was put in place that added greater protection for arrays in heap space, lowering the heap vulnerability measure 10% to 177 MB-seconds. If the ABFT code lengthened running time in such a way that the code vulnerability was increased by 20% to 1.6 MB-seconds, the overall affect on reliability would be positive.

Register vulnerability has not been discussed in these terms for two reasons. First, register memory is inside the CPU and so is not directly comparable to the RAM holding the heap and code. Secondly, register memory is extremely small compared to heap and code memory. The probability of an SEU affecting RAM is orders of magnitude higher than the chance that a register will be affected.

4. Conclusions

The segment_rocks program proved surprisingly robust in the face of faults injected into its heap, registers, and code. In each case, usage profiles illustrate why this is so. This program had particular characteristics that explain its behavior.

The bulk of the heap was in use and vulnerable for only brief periods of time. Most of the time, the program used an array shortly after it was filled with data. Organizing a program so that an array is filled with data as late as possible means that the data has less chance of being corrupted. Our application also had one particularly robust array, called the class array. When a fault injection changed a class array value in the convergence loop, the only effect was to add another iteration to that loop. A final factor was that the data condensation factor in the kmeans part of the code was 48, high enough to give the expectation that most of the time injected faults will not affect the output significantly. The one fault that did produce erroneous output occurred in the very large pointer array that was used. Programs that avoid use of large pointer arrays, (without adversely affecting performance) will do better in these circumstances.

The low register vulnerability also protected this application from being sensitive to register fault injections. Turning off compiler flags to keep this figure low may be an adequate strategy for some programs whose performance is not adversely affected. For programs that need a higher level of register usage, it may be possible to add compiler flags that will keep data from being held unused in the registers for long periods of time.

The lack of program sensitivity to fault injections in the code is due to the structure of the program execution: most of the time is spent at the very end of the program, in a routine that occupies only a small portion of memory.

Calculating the MB-seconds figure for both the code and heap gives a good indication of how vulnerable they are to SEU's. This figure has a number of uses. It can be used to determine whether code or heap space is more vulnerable. It is useful in comparing vulnerability of different programs, or runs of the same program with differing inputs. It can also be useful in evaluating the effectiveness of a particular ABFT strategy. But this vulnerability measure is tied to a particular hardware configuration. To compare figures between different platforms would require a conversion factor.

5. Future Work

This work has been useful in the effort to build a foundation for our project to understand how a program behaves when faults are injected into it. The fault injection efforts described here have been very simple, and limited to heap, registers, and code. No attempts have been made to model SEU's in cache or other areas of the CPU, or in EDAC-protected memory.

It is reasonable to expect that SEU's in cache will have a higher probability of modifying vulnerable areas of memory. In future work we will want to focus on how the program behaves when vulnerable areas are hit. Some explanation of this is detailed here, but more research needs to be done. Such research needs to cover strategies the program can employ to minimize SEU effects. We also want to examine how the vulnerability of the program changes as a function of change in its input data and runtime parameters. We will also look at how the program responds to faults injected into the stack and variable space. Finally, we will want to compare this program's response to fault injections with the response of other programs, to see if the surprising robustness of this program is common in other applications as well.

The MB-seconds figures for code and heap space are not easy to obtain. If it proves to be a useful measure we may want to develop tools that can be used to determine those values.

6. Acknowledgements

This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Appreciation is expressed to Daniel S. Katz and E. Robert Tisdale for their reviews and comments.

7. References

[1] REE Project Overview, http://www-ree.jpl.nasa.gov/overview.html.

[2] D. S. Katz and P. L. Springer, "Development of a Spaceborne Embedded Cluster," to be presented at IEEE International Conference on Cluster Computing (CLUSTER2000), Chemnitz, Germany, November 2000.

[3] M. Turmon and R. Granat, "Algorithm-Based Fault Tolerance for Spaceborne Computing: Basis and Implementations," Proceedings of IEEE Aerospace Conference, IEEE, 2000, Big Sky, Montana.

[4] J. Beahan, L. Edmonds, R. D. Ferraro, A. Johnston, D. S. Katz, and R. R. Some, "Detailed Radiation Fault Modeling of the Remote Exploration and Experimentation (REE) First Generation Testbed Architecture," Proceedings of IEEE Aerospace Conference, IEEE, 2000, Big Sky, Montana.

[5] P. E. Lewkowicz and L. J. Richter, "Single-Event Upsets in Spacecraft Digital Systems," *ISA Transactions*, Vol. 24, No. 4, pp. 45-48.

[6] R. Koga, W. A. Kolasinski, and M. T. Marra, "Techniques of Microprocessor Testing and SEU-Rate Prediction," *IEEE Transactions on Nuclear Science*, Vol. 32, No. 6, December 1985, pp. 4219-4224.

[7] J. H. Elder, J. Osborn, W. A. Kolasinski, "A Method for Characterizing a Microprocessor's Vulnerability to SEU," *IEEE Transactions on Nuclear Science*, Vol. 35, No. 6, December 1988, pp. 1678-1681.

[8] R. D. Rasmussen, "Computing in the Presence of Soft Bit Errors," *Proceedings of American Control Conference*, IEEE, 1984, San Diego, CA, pp. 1125-1130.

[9] V. Asenek, C. Underwood, R. Velazco, S. Rezgui, M. Oldfield, Ph. Cheynet, and R. Ecoffet, "SEU Induced Errors Observed in Microprocessor Systems," *IEEE Transactions on Nuclear Science*, Vol. 45, No. 6, December 1998, pp. 2876-2883.

[10] V. A. Asenek, C. I. Underwood, M. K. Oldfield, "Predicting the Influence of Software on the Reliability of Commercial off-the-shelf (COTS) Technology Microprocessors in a Space Radiation Environment," *Proceedings of AIAA/Utah State University Conference on Small Satellites*, 1997, Logan, UT.